DOI: 10.3745/KIPSTA.2009.16-A.4.245

재겨냥성 C 컴파일러를 위한 테스트 집합 생성 시스템

우 균[†] · 배 정 호[††] · 장 한 일[†††] · 이 윤 정[††††] · 채 흥 석[†††††]

요 약

임베디드 프로세서 사용이 증가함에 따라 임베디드 프로세서를 위한 컴파일러를 시기 적절히 개발해야 할 필요성이 증가하고 있다. 컴파일러 후단부를 수정하여 새로운 컴파일러를 구성하는 재겨냥 기법이 이에 적합한 기법으로 채택되고 있다. 이 논문에서는 재겨냥성 C 컴파일러를 테스트하기 위한 테스트 집합 생성 시스템을 제안한다. 제안한 시스템은 문법 커버리지 개념을 이용하여 테스트 집합을 생성한다. 일반적으로 원시 프로그래밍 언어의 문법을 이용하여 테스트 집합을 생성하면 방대한 크기의 테스트 집합이 얻어진다. 그러나 신속히 컴파일러를 출시해야 하는 상황에서는 방대한 테스트 집합 크기가 문제가 될 수 있다. 이에 이 논문에서 제안한 시스템은 중간 코드를 고려하여 테스트 집합을 축약하는 기능을 탑재하고 있다. 실험 결과에 따르면, 비록 축약된 테스트 집합 크기는 원본 테스트 집합 크기의 평균 10%에 불과하지만 원본 테스트 집합이 검출할 수 있는 컴파일러 오류의 75% 정도를 검출할 수 있음을 알 수 있었다. 이는 본 논문에서 제시한 축약 기법이 임베디드 컴파일러 개발 초기 단계에서 효과적으로 사용될 수 있음을 의미한다.

키워드 : 컴파일러 검사, 재겨냥성 C 컴파일러, 임베디드 시스템, 중간코드

Test Suit Generation System for Retargetable C Compilers

Gyun Woo[†] · Jung-Ho Bae[††] · Han-Il Jang[†††] · Yun-Jung Lee[††††] · Heung-Seok Chae[†††††]

ABSTRACT

With the increasing adoption of embedded processors, the need of developing compilers for the embedded processors with timely manner is also growing. Retargeting has been adopted as a viable approach to constructing new compilers by modifying the back-end of an existing compiler. This paper proposes a test suite generation system for testing retargetable C compilers. The proposed system generates the test suite using the grammar coverage concept. Generally, the size of the test suite satisfying the grammar coverage of the source language is very large. Hence, the proposed system also provides the facility to reduce the size of the test suite. According to the experimental result, the reduced test suite can detect 75% of the compiler faults detected by the original test suite though the size of the reduced test suite is only 10% of that of the original test suite in average. This result indicates that the reduction technique proposed in this paper can be effectively used in the prior phase of the development procedure of the embedded compilers.

Keywords : Compiler Testing, Retargetable C Compiler, Embedded System, Intermediate Code

1. 서 론

임베디드 소프트웨어 사용이 증가함에 따라 많은 임베디드 프로세서가 새로 설계되고 있다. 따라서 새로운 프로세서를 위한 컴파일러 개발 필요성이 증가하고 있는데, 이에 따라 재겨냥성 컴파일러 개발 기법들이 지속적으로 연구되고 있다[1, 2, 3, 4, 5]. 임베디드 시스템에서는 전통적인 컴파일러 기법 외에도 프로세서 아키텍처에 따라 코드 생성 부분만을 변경하는 재겨냥성 컴파일러 기법이 주로 사용되고 있으며, 따라서 재겨냥성 C 컴파일러에 대한 필요성은 매우 높은 실정이다.

재겨냥성 컴파일러(retargeted compiler)란 새로운 프로세서에 적합하도록 기존 컴파일러를 쉽게 변경할 수 있는 컴파일러를 말한다. 프로세서에 독립적인 기존 컴파일러 부분을 그대로 사용할 수 있으므로, 처음부터 새로운 프로세서를 위한 컴파일러를 작성하는 것보다 개발 시간을 훨씬 단축시킬 수 있다. 프로세서에 종속적인 부분을 통상 컴파일러의 후단부(back-ends)라고 부르는데, 재겨냥성 컴파일러는 후단부를 쉽게 변경할 수 있는 컴파일러라고 할 수 있다. 이렇게 후단부를 변경하여 만든 컴파일러를 재겨냥 컴파일

※ 이 논문은 부산대학교 자유과제 학술연구비(2년)에 의하여 연구되었음 (This work was supported for two years by Pusan National University Research Grant)
† 종신회원 : 부산대학교 정보컴퓨터공학부 부교수
†† 준 회 원 : 부산대학교 컴퓨터공학과 박사과정
††† 정 회 원 : 슈어소프트테크 소프트웨어 시험자동화연구소 전임연구원
†††† 정 회 원 : 부산대학교 U-Port 사업단 박사후연구원
††††† 정 회 원 : 부산대학교 정보컴퓨터공학부 조교수
논문접수 : 2009년 6월 4일
수 정 일 : 1차 2009년 7월 6일
심사완료 : 2009년 7월 8일

러, 재목적 컴파일러라고 부르며 재겨냥 컴파일러를 구성하는 것을 재겨냥(retargeting)이라고 부른다. 컴파일러 전단부(front-ends)에서는 원시 프로그램을 중간 코드로 변경하고 후단부에서는 중간 코드를 바탕으로 목적 코드를 생성한다.

재겨냥 기법이 컴파일러 개발 시간을 단축시키는 기법인데도 불구하고 컴파일러 검사 시간을 단축시키는 연구에 대해서는 상대적으로 많은 연구가 진행된 바 없다. 컴파일러를 검사하는 전통적인 방법 중에는 컴파일러 자체를 구현한 소스코드를 다시 컴파일해 보는 방식이 있는데, 다른 컴파일러 검사 방법도 대개 이러한 대형 프로그램을 작성하여 컴파일러가 제대로 동작하는지 검사하고 있는 실정이다. 그러나 재겨냥을 수행한 후에는 목적 프로세서에 종속적인 특정 명령어를 생성하는 패턴이 제대로 컴파일되지 않는 경우가 흔하기 때문에, 대형 프로그램 보다는 작은 테스트 프로그램 여러 개를 수행하는 편이 오류를 탐지하는데 더 효율적이다.

컴파일러를 검사하기 위한 소스 프로그램을 생성하는 방법 중에서 문법 커버리지(grammar coverage) 방법이 있다. 문법 커버리지란 원시 언어(source language)의 문법 규칙을 사용하고 있는가에 대한 기준을 말한다. 어떤 테스트 집합에 있는 프로그램들을 생성하는데 필요한 문법 규칙을 모두 모았을 때, 원시 언어의 문법 규칙 집합을 모두 포함하고 있다면 해당 테스트 집합은 문법 커버리지를 만족한다고 한다. 문법 커버리지를 만족하는 테스트 프로그램은 매우 작은 단위로 구성된 프로그램이므로 재겨냥 컴파일러 개발 초기에 컴파일러를 검사하는데 적합하다고 할 수 있다.

본 논문에서는 재겨냥 컴파일러를 테스트하기 위한 테스트 프로그램 집합 생성 시스템을 제안한다. 제안한 시스템의 중요한 특징으로는 중간코드를 고려하여 테스트 프로그램 집합 크기를 줄일 수 있다는 점이다. 일반적으로 문법 커버리지를 만족하는 테스트 집합을 생성하면 방대한 양의 프로그램 집합이 얻어진다. 프로그래밍 언어의 문법 규칙 자체가 방대한 양으로 구성되어 있기 때문이다. 그러나 재겨냥 컴파일러를 검사하기 위해서 모든 테스트 프로그램이 필요한 것은 아니다. 정작 중요한 것은 재겨냥 과정 중에서 바뀐 부분만을 테스트하면 되기 때문이다.

본 논문의 사전 연구로서 중간코드를 바탕으로 한 재겨냥 컴파일러 테스트 방안이 발표된 바 있다[6, 7]. 본 논문에서는 이 연구의 아이디어를 확장하여 여러 문법 커버리지 기준에 대해 테스트 프로그램이 생성될 수 있도록 하였으며 이 기법을 프로토타입 시스템으로 구현하여 중간코드를 기반으로 한 재겨냥 컴파일러 테스트 기법의 성능을 검사하였다. 검사 방법으로는 뮤테이션 테스트 기법(mutation test)을 이용하였는데 컴파일러를 일부러 변경하여 오류 컴파일러를 만들고 해당 테스트 집합이 이를 검출해 내는지 검사하였다.

본 논문의 구성은 다음과 같다. 2절에서는 문법 커버리지를 기반으로 한 컴파일러 검사 기법을 설명하고 이 논문에서 사용하는 중간 코드인 RTL(register transfer language)에 대해 설명한다. 3절에서는 본 논문의 기본 아이디어를 문법 커버리지 개념을 이용하여 설명한다. 4절에서는 GCC(Gnu C compiler)를 기반으로 구축한 프로토타입 시스템을 설명한다. 5절에서는 뮤테이션 테스트 기법을 통해 테스트 집합의 적합성(adequacy)을 설명하고 6장에서는 관련연구를 기술한다. 끝으로 7장에서 결론을 맺는다.

2. 연구 배경

이 절에서는 문법 커버리지를 기반으로 한 컴파일러 검사 기법과 RTL을 설명한다. RTL은 GCC에서 사용하는 중간 코드인데, 이 논문에서 다루는 재겨냥 컴파일러는 GCC를 기반으로 한 재겨냥 컴파일러다. 따라서 RTL에 대해 간단히 살펴본다.

2.1 문법기반 컴파일러 검사 기법

컴파일러를 검사하기 위해서는 다양한 입력 프로그램에 대해 컴파일러가 올바른 코드를 생성하는지 검사해야 한다. 따라서 이상적으로는 무한한 개수의 소스 프로그램에 대해 검사해야 한다. 그러나 실제로는 동일한 오류를 검사할 수 있는 프로그램은 제외하고 유한한 개수의 소스 프로그램만을 조사하게 되는데, 이러한 테스트 프로그램 집합을 테스트 집합(test suite)이라고 한다.

체계적으로 테스트를 수행하기 위해서 테스트 분야에서는 커버리지(coverage) 개념을 도입하여 사용하고 있다. 커버리지를 정의하기 위해서는 테스트해야 할 대상을 체계적인 단위로 나누는데, 예컨대 프로그램 수행 경로 등이 커버리지 대상이 될 수 있다. 컴파일러의 경우에는 어떤 프로그래밍 언어에 대해서 정의되고 해당 언어의 프로그램은 문법(grammar)에 의해 생성되므로 컴파일러를 테스트할 때에는 자연스럽게 문법을 커버리지 대상으로 삼을 수 있다[8].

문법은 문법 규칙(grammar rules)에 의해 정의되는데, 어떤 테스트 프로그램 집합이 어떤 문법의 모든 규칙을 모두 사용하는 경우에 테스트 프로그램 집합은 문법 규칙을 망라한다(cover)고 하고, 이 때 이 테스트 프로그램 집합을 문법 규칙 커버리지(grammar rule coverage)라고 부른다. 다시 말해서, 문법 규칙 커버리지가 만족되는 테스트 프로그램 집합에는 모든 문법 규칙에 대해 그 규칙을 사용하여 생성된 프로그램이 존재한다는 의미가 된다. 문법 규칙 커버리지는 문법을 기반으로 한 소프트웨어를 테스트하는데 필수적인 요건으로 받아들여지고 있다[9, 10, 11].

문법 규칙 커버리지는 두 가지로 분류할 수 있는데, 하나는 Purdom이 제안한 커버리지 방식[10]이며 다른 하나는 n-상태 경로 커버리지 방식[13]이다. Purdom이 제안한 방식은 문법의 시작 기호(start symbol)부터 시작하여 비단말 기호(nonterminals)를 차례로 지워나가는 방식이다. 비단말 기호가 더 이상 존재하지 않게 되었을 때 생성된 프로그램을 테스트 프로그램으로 사용한다. 모든 문법 규칙이 적어도 한 번 이상 사용될 때까지 이 과정을 반복한다.

n-상태 경로 커버리지 방식은 유도 과정(derivation sequence)을 트리 유한 상태 기계(tree finite state machine)로 해석한다. 이렇게 변환하게 되면 모든 비단말 기호는 상태로 변환되고 유도 과정은 유한 상태 기계의 경로로 해석할 수 있다. 1-상태 경로 커버리지란 모든 상태가 적어도 한 번 이상 거쳐지는 프로그램을 의미한다. 2-상태 경로 커버리지는 모든 가능한 두 개의 상태 조합이 적어도 한 번 이상 사용되는 프로그램을 의미한다.

2.2 RTL

RTL은 GCC[1]의 중간 코드로서 GCC의 전단부가 수행된 후에 생성되는 코드다. GCC에서 RTL의 위치는 (그림 1)과 같다. 따라서 GCC를 기반으로 재겨냥 C 컴파일러를 생성하기 위해서는 RTL에서 기계어 코드를 생성하는 후단부만 변경해 주면 된다.

RTL 코드는 RTL 표현식(RTL expression)으로 구성된다. 각 RTL 표현식은 이중 연결 리스트 형태로 연결되어 있는데 RTL 수식에는 RTL 객체가 포함되어 있다. RTL 객체 중에서 특정 부류가 목적 기계의 동작을 나타내는 코드 역할을 한다. RTL 코드의 예를 들면 (그림 2)와 같다.

(그림 2)의 RTL 코드는 대입 명령어(assignment instruction)를 나타낸다. insn 다음에 있는 숫자 세 개는 각각 RTL 명령어 번호와 이전 명령어, 다음 명령어 링크를 나타낸다. 여기서 명령어 이름은 set인데 set은 대입 명령어에 해당한다. 이기서 RTL 명령어는 레지스터 내용과 정수 상수를 논리합(logical and)한 결과를 다시 그 레지스터에 저장하는 명령어다.

RTL 표현식으로부터 목적 기계어를 생성하기 위해서 RTL 패턴을 사용한다. GCC를 바탕으로 재겨냥 컴파일러를 작성할 때에는 새로운 기계에 적합한 RTL 패턴을 작성하는 것이 주요 작업이 된다. RTL 패턴을 작성하고 나면 GCC 후단부는 RTL 패턴과 기계 기술서(machine description)로부터 자동으로 생성된다.

(그림 1) GCC에서 RTL의 위치

```
(insn 5 2 6
  (set (reg/f:SI 7 sp)
       (and:SI (reg/f:SI 7 sp)
               (const_int -16 [0xfffffff0])))
  -1 (nil) (nil))
```

(그림 2) RTL 코드 예

1) GCC는 Gnu Compiler Collection이라는 뜻도 있는데, 이 경우에는 여러 컴파일러를 합쳐서 GCC라고 부른다. 여기서는 Gnu C Compiler라는 뜻으로 사용하겠다.

3. 기본 아이디어

본 논문에서 제안하는 시스템의 기본 아이디어는 중간 코드를 고려하여 테스트 집합을 생성해야 한다는 것이다. 그러나 처음부터 중간 코드만을 고려하여 테스트 집합을 생성하지는 않는다. 테스트 집합은 두 단계를 거쳐 생성되는데 첫 번째 단계에서는 프로그래밍 언어의 문법에 따라 테스트 프로그램들을 생성하며 두 번째 단계에서는 중간 코드의 문법을 기준으로 테스트 프로그램들을 선별한다. 각 단계에서 앞서 설명한 문법 규칙 커버리지 개념을 사용한다.

3.1 원시 언어 문법 커버리지

컴파일러와 같이 문법을 기초로 하여 작성되는 프로그램에서는 문법을 기초로 테스트 입력을 구성하는 것이 자연스럽다. 블랙박스 검사를 가정한다면 입력 명세에 따라 입력을 구성하는 것이 바람직한데, 컴파일러의 입력 명세는 프로그래밍 언어의 문법이다. 따라서 모든 문법 규칙을 사용하도록 테스트 프로그램을 작성하는 것이 바람직하다.

C 컴파일러를 기준으로 간단한 예를 살펴보자. (그림 3)(a)는 테스트 C 프로그램을 나타낸다. 이 프로그램에는 덧셈 연산자가 포함되어 있다. C 문법 중에서 덧셈 연산자를 포함하고 있는 문법 규칙은 (그림 3)(b)뿐인데, 따라서 (그림 3)(a) 프로그램을 생성하기 위해서는 반드시 (그림 3)(b) 문법규칙을 사용해야 한다. 이 때 (그림 3)(a) 프로그램은 (그림 3)(b) 규칙을 망라한다고 한다.

원시 언어의 문법 커버리지는 컴파일러를 테스트하기 위한 최소한의 요구사항이라고 볼 수 있다. 검사 강도를 높이기 위해서, 그래서 더 많은 오류를 검사하기 위해서는 문법 커버리지의 깊이를 고려해 볼 수 있다. 지금까지는 모든 문법 규칙이이 한 번만 사용되는 경우를 가정하였지만, 모든 문법 규칙이 두 번 이상 사용되는 경우를 생각해 볼 수도 있다. 좀 더 높은 강도로 테스트를 수행하기 위해서는 문법 커버리지의 깊이를 고려하여 테스트 집합을 생성할 수 있다.

3.2 RTL 문법 커버리지

이 논문의 기본 아이디어는 중간 코드 RTL을 고려하여 더 작은 테스트 집합을 생성하는 것이다. 컴파일러 전체를

```
int f(int a) {
    return a + a;
}
```

(a) 테스트 프로그램

additive-expression:
addtive-expression + *multiplicative-expression*

(b) 사용되는 문법 규칙

(그림 3) 테스트 C 프로그램과 사용되는 문법 규칙 예

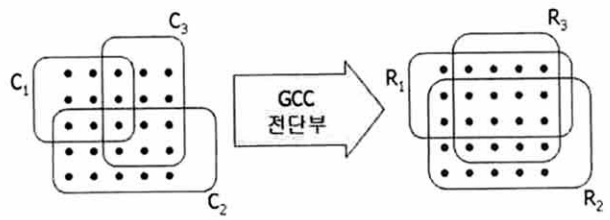
검사하기 위해서는 원시 언어 문법 커버리지를 고려해야 하겠지만, 컴파일러 후단부만 검사한다면 중간 코드에 해당하는 문법의 커버리지를 고려하는 것이 더 적합하기 때문이다. 실제로 재겨냥 과정을 통해 생성된 컴파일러는 후단부만을 새로 작성한 컴파일러이므로 이 방법이 더 효율적이다.

원시 언어 대신 중간 코드 RTL를 고려한다고 해서 RTL 프로그램을 생성한다는 의미는 아니다. 다만 원시 언어를 고려하여 생성된 테스트 집합 중에서 RTL 문법 커버리지를 고려하였을 때 불필요하다고 생각되는 프로그램을 제거하는 것이다. 따라서 RTL을 고려하여 축약된 테스트 집합도 여전히 원시 언어로 작성된 테스트 프로그램의 집합이 된다. (그림 4)는 원시 언어의 문법 커버리지와 RTL 문법 커버리지의 차이점을 보여준다.

(그림 4)에서 점들은 문법 규칙을 나타낸다. 왼편에 있는 검은 점들은 C 문법 규칙을 나타내고 오른편에 있는 회색 점들은 RTL 문법 규칙을 나타낸다. 둥근 네모는 테스트 프로그램을 나타낸다. C_i는 C 프로그램을 나타내고 R_i는 C_i에 대응하는 RTL 프로그램을 나타낸다. 각 C 프로그램은 GCC 전단부를 거치면 RTL 프로그램으로 변환된다.

(그림 4)에서 볼 수 있는 것처럼 C 문법 커버리지를 위해 필요했던 프로그램이 RTL 문법 커버리지를 위해서는 불필요할 수 있다. (그림 4)에는 세 개의 C 프로그램이 있는데 이 프로그램은 C 문법을 모두 망라하기 위해서 모두 필수적인 프로그램들이다. 그러나 전단부를 거쳐 RTL 프로그램들이 생성되면 상황이 달라질 수 있다. (그림 4)의 오른편에서 볼 수 있는 것처럼 R_3에 의해 망라되는 RTL 문법 규칙은 이미 R_1과 R_2에 의해 망라되고 있으므로 R_3은 RTL 커버리지를 위해서 필수적인 프로그램이 아니다. 따라서 R_3에 해당하는 원시 프로그램 C_3은 RTL 커버리지 관점에서는 불필요한 프로그램이다.

RTL 문법 커버리지를 만족한다고 해서 원시 언어의 문법 커버리지를 만족한다는 의미는 아니다. RTL 문법 커버리지를 고려하는 이유는 재겨냥성 컴파일러의 특징을 고려하여 원본 테스트 집합 크기를 줄이기 위해서다. 따라서 RTL 문법 커버리지를 고려하여 생성된 테스트 집합은 항상 원본 테스트 집합의 부분집합이며 따라서 원시 언어의 문법 커버리지를 만족하지 않을 수 있다. 다만 이렇게 축약된 테스트 집합이 재겨냥성 컴파일러 검사에 관한 한, 더 높은 효율을 보일 수 있다는 것이 RTL 문법 커버리지의 의미라고 볼 수 있다.



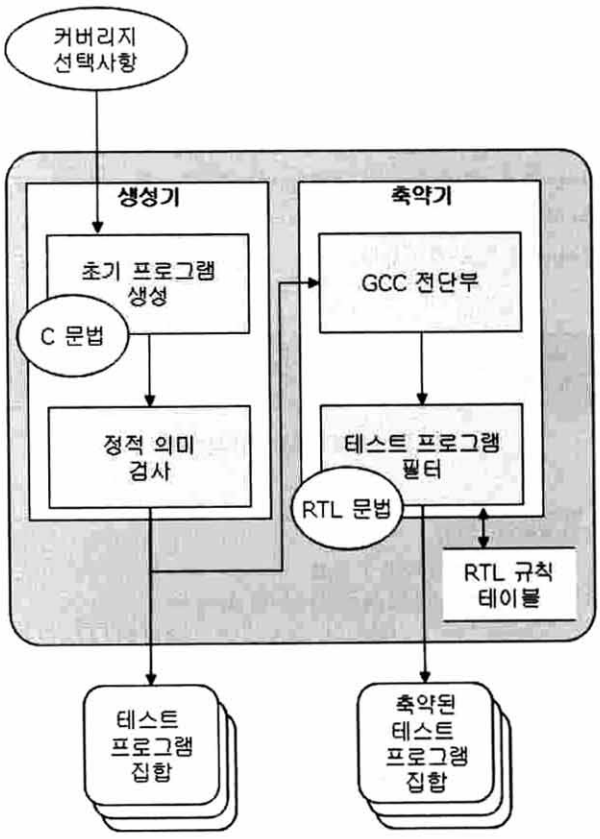
(그림 4) C 문법 커버리지와 RTL 문법 커버리지

4. RTL 기반 테스트 프로그램 생성기 구현

이상에서 기술한 아이디어를 바탕으로, RTL을 기반으로 하여 C 컴파일러에 대한 테스트 프로그램을 생성하는 시스템을 구현하였다. 시스템 구성도를 나타내면 (그림 5)와 같다. 이 시스템의 입력으로는 C 문법 커버리지 선택사항이고 출력은 생성된 테스트 프로그램 집합과 축약된 테스트 프로그램 집합이다. 실제 시스템 출력은 축약된 테스트 프로그램 집합뿐이지만 두 테스트 프로그램 집합의 적합성을 비교하기 위해서 두 집합을 모두 출력으로 제시하도록 하였다.

(그림 5)에서 볼 수 있는 바와 같이 시스템은 크게 두 부분으로 구성된다. 하나는 테스트 프로그램 생성기이며 다른 하나는 테스트 프로그램 축약기다. 생성기에서는 C 문법을 기준으로 C 프로그램을 생성한다. 생성된 C 프로그램은 C 언어의 정적 의미(static semantics)에 맞지 않을 수 있기 때문에 정적 의미를 검사하여 컴파일 가능한 프로그램만을 생성하도록 한다.

축약기는 생성된 프로그램 집합에서 RTL 문법 규칙을 망라하는 프로그램만을 선별하여 테스트 프로그램 집합을 축약한다. RTL 코드를 생성하기 위해서 GCC 전단부를 이용하였다. RTL 문법 규칙 커버리지를 검사하기 위해 RTL 문법을 사용하고 있으며 각 프로그램마다 어떤 RTL 규칙을 커버하는지 RTL 규칙 테이블을 갱신해 가며 테스트 프로그램 집합을 축약한다.



(그림 5) RTL 기반 테스트 프로그램 생성기 구조

C 문법으로부터 테스트 프로그램을 생성할 때 선택사항으로서 어떤 기준을 사용할 것인가 결정할 수 있다. 현재 시스템에서 지원하는 선택사항으로는 다음 세 가지가 있다.

- P1(Purdom의 알고리즘): 모든 문법 규칙이 적어도 1회 이상 사용되도록 함
- C1(1-상태 경로 커버리지): 모든 문법 기호가 적어도 하나의 프로그램 경로에서 사용됨. 각 문법 규칙은 1회 이상 사용됨
- C2(2-상태 경로 커버리지): 가능한 모든 문법 기호를 2개 조합한 것이 적어도 하나의 프로그램 경로에서 사용됨. 각 문법 규칙은 2회 이상 사용됨

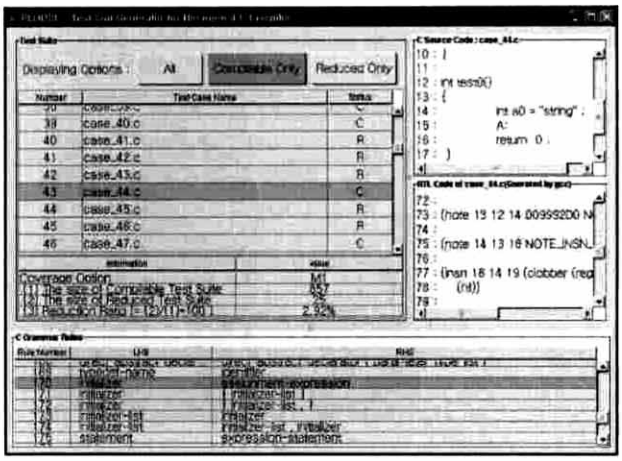
Purdom의 알고리즘을 이용하면 각 문법 규칙을 사용하는 가작 작은 테스트 프로그램 집합을 생성할 수 있다. 간혹 Purdom의 알고리즘으로 생성된 프로그램은 정적 의미 규칙을 준수하지 않는 경우가 있었는데 이러한 형태의 프로그램은 수동으로 보정하는 것이 필요하다.

Purdom의 알고리즘과 달리 n-상태 경로 커버리지 알고리즘은 수동 보정이 필요 없다. n-상태 경로 커버리지 알고리즘에서 C1을 선택하면 모든 문법 규칙이 최소 1회씩 사용되도록 테스트 프로그램 집합이 생성되며, C2를 선택하면 조합 가능한 모든 문법 규칙이 최소 2회씩 사용되도록 테스트 프로그램이 생성된다.

(그림 5)에서 보인 것처럼 테스트 프로그램 생성기는 두 개 모듈로 구성되는데, 하나는 초기 테스트 프로그램 생성 모듈이며 다른 하나는 정적 의미 검사 모듈이다. 초기 테스트 프로그램 생성 모듈에서는 문법만을 고려하여 테스트 프로그램을 생성한다. 이 과정에서는 프로그램의 정적 의미를 고려하지 않고 문법만을 고려하기 때문에 컴파일되지 않는 프로그램을 생성할 수도 있다. 따라서 정적 의미 검사 루틴에서 컴파일되는 프로그램만을 선별하고 이에 따라 커버되지 않는 문법 규칙을 커버하는 테스트 프로그램은 다시 생성되도록 한다.

테스트 프로그램 축약기의 역할은 컴파일 가능한 테스트 프로그램 집합을 축약하여 더 작은 테스트 프로그램 집합을 생성하는 것이다. 앞서 언급한 것처럼 테스트 프로그램을 축약하는 기준은 RTL 커버리지다. 테스트 프로그램 축약기에서는 각 테스트 프로그램에 대한 RTL 코드 생성하기 위해 GCC 전단부를 이용한다. 생성된 RTL 코드에서 커버되는 RTL을 판단하기 위해서 별도의 테스트 프로그램 필터를 사용한다.

RTL 커버리지를 고려하여 테스트 프로그램 집합을 결정하는 방법은 여러 가지가 있다. 우리가 원하는 것은 최적의 커버리지가 되도록 생성하는 것이지만 최적 커버리지 문제는 NP-완전 문제이기 때문에 구현한 테스트 프로그램 필터에서는 간단한 순차 알고리즘을 사용하고 있다. 5장에서 설명하겠지만, 간단한 순차 알고리즘만 선택해도 성능은 큰 문제가 없는 것으로 드러났다. 현재 구현된 프로토타입 시스템은 (그림 6)과 같다.



(그림 6) RTL 기반 테스트 프로그램 생성 프로토타입 시스템

현재 이 프로토타입 시스템은 Linux 상에서 자바 1.5를 이용하여 작성되었다. GCC를 Java에서 구동시킬 수 있어야 하기 때문에 Linux를 선택하였지만, GCC가 지원되는 다른 시스템에도 이식시킬 수 있으리라 생각한다. 이 시스템을 구현할 때 필요한 문법은 두 가지가 있는데, 하나는 원시 언어 문법(C 문법)이고 다른 하나는 중간 언어 문법(RTL 문법)이다. C 문법으로는 표준 ISO C 문법을 사용하였고 RTL 문법은 독자적으로 정의하여 사용하였다. (그림 6)의 왼편에서는 생성된 테스트 프로그램 집합과 해당 정보를 볼 수 있으며 오른편에서는 선택된 테스트 프로그램에 대한 C 코드와 RTL 코드를 볼 수 있다. 아래 부분에서는 선택된 프로그램에 대하여 커버되는 C 문법 규칙을 보여주고 있다.

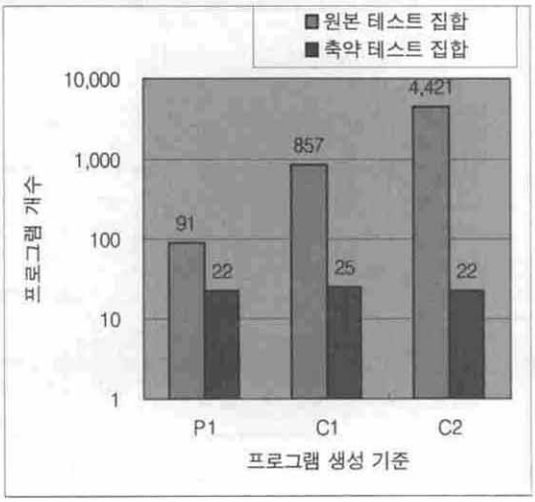
5. 실 험

이 논문에서 제시한 방법론의 성능과 효과를 검증하기 위해 구현한 프로토타입 시스템을 이용하여 실험을 시행하였다. 이 실험을 통해 입증하고자 하는 가설은 다음 두 가지이다.

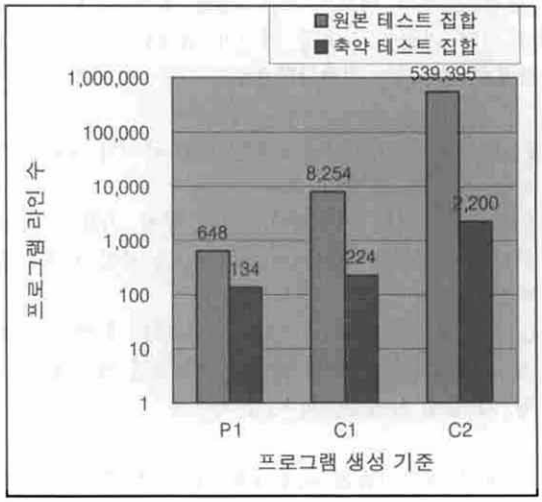
- 중간 코드를 이용한 커버리지 방법을 이용하면 실제로 테스트 프로그램 집합 크기를 줄일 수 있음
- 테스트 프로그램 집합의 오류 검출 능력이 급격히 감소하지 않음

위 두 가설 중 첫 번째 가설은 쉽게 입증할 수 있다. 테스트 집합 생성기 모듈에서 일차적으로 생성된 테스트 프로그램 집합과 축약기 모듈을 거쳐 최종적으로 생성된 테스트 프로그램 집합의 크기를 비교해 보면 되기 때문이다. 중간 코드를 이용한 테스트 집합 축약 효율을 그림으로 나타내면 (그림 7)과 같다.

테스트 프로그램 집합에 대한 축약 효율은 프로그램 개수



(a) 테스트 프로그램 개수 축약률



(b) 테스트 프로그램 라인 수 축약률

(그림 7) 중간 코드를 이용한 테스트 프로그램 집합 축약 효율

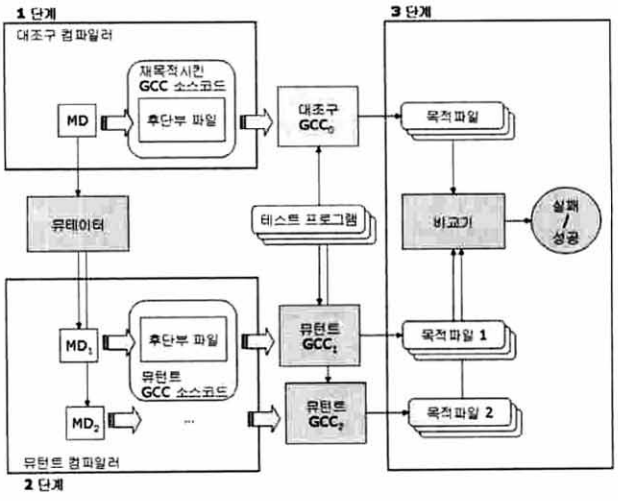
와 프로그램 라인 수, 두 가지 측면에서 측정하였다. (그림 7)(a)는 테스트 프로그램 개수 축약률을 나타내고 (그림 7)(b)는 테스트 프로그램 개수 축약률을 나타낸다. (그림 7)에서 y축은 각각 테스트 집합의 총 프로그램 개수와 총 프로그램 라인 수를 나타내는데, 모두 로그 눈금 간격임에 주의하자. 테스트 프로그램 생성 기준 P1, C1, C2 각각에 대하여, 프로그램 개수 측면에서는 75.8%, 97.1%, 99.5%의 축약률을 보였으며 평균 90.1% 축약률을 보였다. 라인 수 측면에서는 P1, C1, C2 각각 79.3%, 97.3%, 99.6%의 축약률을 보였으며 평균 91.6% 축약률을 보였다.

위 결과에 따르면 테스트 프로그램 축약률이 상당히 높음을 알 수 있다. 그러나 이렇게 축약함으로 인해서 테스트 프로그램 집합의 오류 검출 능력, 즉 테스트 집합으로서의 적합성(adequacy)을 현저히 떨어뜨리는 것이 아닌가 하는 의문이 생긴다. 테스트 집합의 적합성을 검사하기 위해 뮤테이션 테스트 기법[14, 15]을 이용하여 각 테스트 집합의 적합성에 대하여 실험하였다.

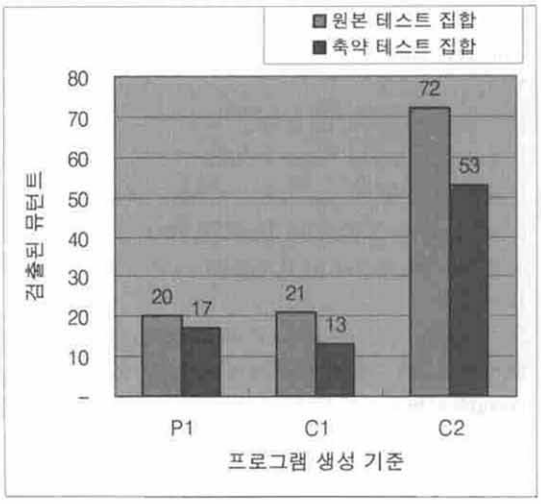
뮤테이션 테스트[14, 15]란 테스트 집합의 적합성을 검증하기 위한 방법이다. 테스트 집합의 성능을 평가하기 위해 테스트 대상에 일부러 오류를 삽입한 뮤턴트(mutant)를 여러 개 생성한다. 뮤턴트는 오류가 삽입된 것이기 때문에 테스트 집합은 이들 뮤턴트가 오류임을 검증해 내야 한다. 테스트 집합에 대해 올바르게 수행되지 않는 뮤턴트를 실패한 뮤턴트(killed mutant)라고 하는데, 테스트 집합의 적합성은 전체 뮤턴트 개수에 대한 실패한 뮤턴트 개수의 비율로 나타낼 수 있다.

우리가 테스트하고자 하는 대상은 컴파일러이므로 컴파일러를 뮤턴트로 생성해야 한다. GCC를 재겨냥시킬 때에는 기계 명세서(machine description: MD) 파일과 타겟 매크로(target macro) 파일을 변경하여 컴파일러를 구성해야 한다. 이 실험에서는 편의상 타겟 매크로는 고려하지 않고 기계 명세서 파일만을 변경하여 후단부가 변경된 뮤턴트 GCC를 생성하였다. 뮤턴트 GCC를 생성한 후에는 테스트 프로그램 집합을 원본 GCC(대조구 GCC)와 뮤턴트 GCC로 모두 컴파일한 후에 그 결과 파일을 비교하여 뮤턴트 GCC를 테스트 집합이 검출해 내는지 판단하였다. 이 과정을 그림으로 나타내면 (그림 8)과 같다.

(그림 8)에 보인 뮤테이션 테스트 단계는 크게 세 단계로 구성되어 있는데, 1단계는 대조구 컴파일러(reference) 컴파일러를 구성하는 단계이며 2단계는 뮤턴트 컴파일러를 구성하는 단계다. 대조구 컴파일러로는 Intel x86에 대한 GCC를 사용하였다. 목적 기계 기술서(MD)의 여러 부분을 하나씩 변경하여 뮤턴트 컴파일러를 생성하므로, 뮤턴트 컴파일러는 여러 개가 생성된다. 3단계는 대조구 컴파일러와 뮤턴트 컴파일러 각각에 대해 테스트 프로그램 집합의 모든 프로그램을 입력하여 출력으로 생성된 목적파일이 같은지 비교하



(그림 8) GCC 뮤테이션 테스트 과정



(그림 9) 뮤테이션 테스트 결과

는 단계다. 이 단계에서 해당 실패한 뮤턴트(killed mutant)를 가려낸다. 뮤테이션 테스트 결과를 나타내면 (그림 9)와 같다.

적합성 실험 결과에 따르면 아쉽게도 두 테스트 집합의 적합성이 다르다. P1, C1, C2 중에서 C2가 가장 많은 뮤턴트를 검출해 냈는데 각각 20, 21, 72개의 뮤턴트를 검출해 냈다. 반면 축약된 테스트 집합은 P1, C1, C2 각각의 경우에 17, 13, 53개의 뮤턴트를 검출해 냈다. 원본 테스트 집합에 대하여 축약된 테스트 집합의 뮤턴트 검출 비율은 P1, C1, C2의 경우 각각 15.0%, 38.1%, 26.4% 감소하였으며, 평균 24.7% 감소하였다. 뮤턴트 검출 비율이 감소한 것은 명백한 사실이지만 원본 테스트 집합의 10% 정도의 크기의 테스트 집합을 이용하여 원본 테스트 집합이 검출할 수 있는 컴파일러 오류의 75% 정도를 검출할 수 있다는 것은 테스트 효율 측면에서 긍정적이라고 평가할 수 있다.

테스트 시간 측면에서 살펴볼 때, 원본 테스트 집합을 이용하여 뮤테이션 테스트 스크립트를 수행하는 데 실제로 20시간 넘게 걸렸다. 그러나 축약된 테스트 집합은 원본 테스트 집합 크기의 10% 정도이므로 축약된 테스트 집합을 이용하면 이 시간이 2시간 내외로 단축된다는 의미가 된다. 이는 스크립트를 이용한 테스트의 경우이므로, 실제로 테스터가 개입될 경우에는 테스트 시간이 더 늘어날 것으로 예상된다. 따라서 본 논문의 기법을 이용하면 테스트 시간을 효과적으로 단축시킬 수 있을 것이라고 전망된다.

뮤테이션 테스트 결과, 원본 테스트 집합과 축약된 테스트 집합 모두 모든 컴파일러 오류를 검출하지는 못했다. 생성된 총 뮤턴트 개수가 573개임을 감안할 때, 가장 많은 오류를 검출한 테스트 집합 C2도 72개의 뮤턴트만 검출해 내는 데 그쳤다. 검출률로 본다면 12.6%에 불과한 수치다. 이는 컴파일러 오류가 단순히 문법 커버리지 측면에서만 발생하는 것이 아니라는 사실을 보여준다. 사실 프로그래밍 언어 이론 관점에서는 구문 규칙에 해당하는 문법 외에도 정적 의미론, 동적 의미론 등 의미론적 측면이 프로그래밍 언어를 규정하는 데 중요한 역할을 한다고 볼 수 있다. 따라서 이런 의미론을 망라할 수 있는 테스트 방법을 고려해야만 보다 넓은 범위의 오류를 검출할 수 있는 테스트 집합을 생성할 수 있을 것이다.

6. 관련 연구

컴파일러 검사와 관련한 연구로는 몇 가지가 있다. Boujarwah와 Saleh는 다양한 컴파일러 검사 기법을 문법 종류, 데이터 정의 커버리지(data definition coverage), 구문 커버리지(syntax coverage), 의미 커버리지(semantic coverage) 등 여러 측면에서 비교하였다[8]. Kossatchev and Posypkin은 프로그래밍 언어의 구문론과 의미론의 엄밀한 명세를 기반으로 하여 테스트 케이스 집합의 생성, 실행, 검사 등을 설명하였다[16].

프로그래밍 언어의 문법을 기초로 하여 컴파일러를 테스트하는 방법에 대한 연구도 진행된 바 있다[17, 18, 19], 이 방법론은 속성 문법(attribute grammar)과 커버리지를 기반으로 한, 테스트 케이스 생성 방법을 제시한다. Zelenov 등은 문법 변환(grammar transformation)을 기초로 한, 컴파일러 및 텍스트 처리 소프트웨어의 테스트 방법을 제시하였다[20].

Kalinov 등은 추상 상태 기계를 기반으로 한, 컴파일러 테스트 방법을 제시하였다[13, 21, 22]. 이 방법은 구문론뿐만 아니라 의미론도 고려하여 테스트 케이스 생성하였는데, 의미론은 Montage라는 상태 기계를 통해 정의하였다. Kalinov 등은 이 방법을 이용하여 mpC라는 언어에 대해 커버리지 기준을 제시하였다. 이 외에도 컴파일러 검사와 관련한 오러클(oracle) 구현 문제에 대한 연구도 진행된 바 있다[23, 24, 25].

최근 Hennessy와 Power는 문법을 기반으로 한 소프트웨어를 테스트하기 위한 최소 테스트 집합을 생성하는 방법을 제시한 바 있다[26]. 이들은 원래 Jones 등이 제시한 테스트 집합 축약 방법[27]을 적용하여 기본적인 규칙 커버리지를 만족하는 테스트 집합을 축약하였다.

앞서 제시한 방법들은 모두 원시 언어의 문법을 기초로 한 방법들이다. 이 논문에서 제시한 방법은 원시 언어 대신 중간 언어의 문법을 고려하여 테스트 집합을 줄이는 방법이다. 이 논문의 선행 연구로서 Purdom 알고리즘을 이용하여 테스트 프로그램들을 생성하고 중간 언어의 문법을 고려하여 테스트 집합을 줄이는 아이디어를 발표한 바 있다. 이 논문에서는 여기에 더하여 n-경로 커버리지 기준을 추가하였으며 이 아이디어를 독립적인 시스템으로 제시하였다.

또한 축약된 테스트 집합의 오류 검출 효과를 검증하기 위해 뮤테이션 테스트를 활용하였다. 뮤테이션 테스트 결과에 따르면 축약된 테스트 집합의 오류 검출 능력이 평균 24.7% 감소한 것으로 드러났다. 그러나 테스트 집합의 축약 비율이 90.1%이라는 점을 감안하면 오류 검출 능력의 저하는 비교적 높지 않음을 알 수 있다. 따라서 본 연구는 재겨냥 컴파일러를 신속히 테스트해야 하는 상황에서 큰 도움이

될 것이라고 생각한다. 엄밀한 테스트가 필요한 경우에는 여전히 축약 이전의 테스트 집합을 이용할 수 있다.

7. 결 론

이 논문에서는 재겨냥성 C 컴파일러를 검사하기 위한 테스트 집합을 생성하는 시스템을 제시하였다. 구현된 시스템은 GCC를 기반으로 하고 있는데, 테스트 집합은 C 문법을 기준으로 모든 문법 규칙을 커버하도록 생성하였으며 중간 코드인 RTL을 고려하여 테스트 집합을 축약하였다. 재겨냥 컴파일러는 후단부만 변경한 컴파일러이므로 원시 언어의 문법 규칙 보다는 중간 언어의 문법 규칙을 고려하는 것이 더 자연스럽다. 실험 결과에 따르면 테스트 집합의 크기는 90% 정도 축약됨을 알 수 있었다. 축약된 테스트 집합의 오류 검출 능력은 평균 24.7% 감소하였지만 이는 테스트 집합 축약률에 비하면 낮은 수치이므로, 초기 단계에서 신속히 컴파일러를 검사해야 하는 경우에 효과적으로 사용될 수 있을 것이라고 생각한다.

현재 구현된 시스템은 GCC의 RTL을 기준으로 작성되었지만, 본 논문의 아이디어는 다른 중간 코드를 사용하는 컴파일러에도 적용될 수 있다. 또한 RTL을 기준으로 하고 있기 때문에, C 컴파일러뿐만 아니라 Fortran, Ada 등 다른 언어의 컴파일러에도 적용할 수 있다. 현재 C 언어를 선택한 이유는 재겨냥 컴파일러로 가장 많이 개발되는 컴파일러가 C 컴파일러이기 때문이다. 이 논문에서 제시한 아이디어는 C 언어뿐만 아니라 다른 언어에도 적용될 수 있다.

논문의 실험 결과에 따르면 축약된 테스트 집합의 오류 검출 능력이 25% 가량 감소하였는데, 오류 검출 능력을 유지하기 위한 방법을 연구하는 것을 향후 연구로 생각해 볼 수 있다. 현재로서는 중간 코드 문법만 고려하고 있는데, 중간 코드 문법 외에 코드 생성 패턴의 의미 등을 고려한다면 테스트 집합의 축약률이 다소 저하될 수는 있겠지만 오류 검출 능력을 유지하는데 도움이 될 것이라고 생각한다.

또한 논문의 실험 결과가 재겨냥성 컴파일러에 일반적으로 적용된다는 것을 입증하기 위해서 보다 다양한 목적 기계를 대상으로 실험을 수행할 필요가 있다. 현재 ARM과 TI 등을 대상으로 실험을 확대하는 것을 고려하고 있는데, 더 넓은 범위의 목적 기계를 대상으로 테스트 집합의 오류 검출 능력을 검사해 보는 것도 향후 연구로 생각해 볼 수 있다.

참 고 문 헌

[1] P. Marwedel. Tree-Based Mapping of Algorithms to Predefined Structures. In IEEE/ACM International Conference on CAD, pp.586-593, 1993.

[2] R. Leupers and P. Marwedel. Retargetable Generation of Code Selectors from HDL Processor Models. In European Design and Test Conference, pp.140-144, 1997.

[3] G. Araujo, S. Malik, and M. T.-C. Lee. Using Register-Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures. In Design Automation Conference, pp.591-596, 1996.

[4] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. Chess: Retargetable Code Generation for Embedded DSP Processors, In Proceedings on Code Generation for Embedded Processors, pp.85-102, 1994.

[5] S. Hanono, S. Devadas. Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator, In Proceedings of the 35th annual conference on Design automation, pp.510-515, 1998.

[6] 장한일, 우균, 채흥석, 중간표현을 이용한 재목적 컴파일러의 효율적인 테스트 방법, 한국정보과학회 2006 가을 학술발표논문집 제33권 제2호(B), pp.575-579, 2006.

[7] Gyun Woo, Heung Seok Chae, and Hanil Jang, An Intermediate Representation Approach to Reducing Test Suites for Retargeted Compilers, In Proceedings of 12th Ada-Europe International Conference on Reliable Software Technologies, pp.100-113, 2007.

[6] A. S. Boujarwah and K. Saleh. Compiler Test Case Generation Methods: a Survey and Assessment. Information and Software Technology, 39(9): 617-625, 1997.

[7] Boris Beizer. Software Testing Techniques. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[8] A. Gargantini and E. Riccobene. ASM-based Testing: Coverage Criteria and Automatic Test Sequence. Journal of Universal Computer Science, 7(11): 1050-1067, 2001.

[9] Keith W. Miller. A Modest Proposal for Software Testing. IEEE Software, 18(2): 98-100, 2001.

[10] P. Purdom. A Sentence Generator for Testing Parsers. BIT Numerical Mathematics, 12(3): 366-375, 1972.

[11] Alexey Kalinov, Alexander Kossatchev, Alexandre Petrenko, Mikhail Posypkin, and Vladimir Shishkov. Coverage-Driven Automated Compiler Test Suite Generation. Electronic Notes Theoretical Computational Science, 82(3), 2003.

[12] R. A. DeMillo, R. J. Lipton, and F.G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer, 11(4): 34-41, 1978.

[13] Richard G. Hamlet. Testing Programs with the Aid of a Compiler. IEEE Transactions on Software Engineering, 3(4): 279-290, 1977.

[14] Alexander Kossatchev and Mikhail Posypkin. Survey of Compiler Testing Methods. Programming and Computer Software, 31(1):10-19, 2005.

[15] John Hannan and Frank Pfenning. Compiler Verification in LF. In Andre Scedrov, editor, In Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science, pp.407-418. 1992.

[16] Jorg Harm and Ralf Lämmel. Two-Dimensional Approximation

Coverage. Informatica (Slovenia), 24(3), 2000.
[17] Ralf Lämmel. Grammar Testing. In Proceedings on Fundamental Approaches to Software Engineering (FASE) 2001, volume 2029 of LNCS, pp.201-216. 2001.
[18] Sergey V. Zelenov, Sophia A. Zelenova, Alexander Kossatchev, and Alexandre Petrenko. Test Generation for Compilers and Other Formal Text Processors. Programming and Computer Software, 29(2): 104-111, 2003.
[19] A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, and V. Shishkov. Using ASM Specifications for Compiler Testing, Lecture Notes in Computer Science, volume 2589, p. 415, 2003.
[20] A. Kalinov, A. Kossatchev, M. Posypkin, and V. Shishkov. Using ASM Specifiation for Automatic Test Suite Generation for mpC Parallel Programming Language Compiler, In Proceedings of the Fourth International Workshop on Action Semantics, pp.99-109, 2002.
[21] Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. Comparison Checking: An Approach to Avoid Debugging of Optimized Code. In ESEC / SIGSOFT FSE, pp. 268-284, 1999.
[22] Tim S. McNerney. Verifying the Correctness of Compiler Transformations on Basic Blocks Using Abstract Interpretation. SIGPLAN Notices, 26(9): 106-115, June, 1991.
[23] George C. Necula. Translation Validation for an Optimizing Compiler. ACM SIGPLAN Notices, 35(5): 83-94, 2000.
[24] Mark Hennessy and James F. Power. An Analysis of Rule Coverage as a Criterion in Generating Minimal Test Suites for Grammar-Based Software. In ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pp.104-113, 2005.
[25] James A. Jones and Mary Jean Harrold. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. IEEE Transactions on Software Engineering, 29(3): 195-209, 2003.

우 균

e-mail : woogyun@pusan.ac.kr
1991년 한국과학기술원 전산학(학사)
1993년 한국과학기술원 전산학(석사)
2000년 한국과학기술원 전산학(박사)
2009년 현 재 부산대학교 정보컴퓨터공학부 부교수
관심분야 : 프로그래밍언어 및 컴파일러, 함수형 언어, 그리드컴퓨팅, 소프트웨어 메트릭, 소프트웨어 테스팅 등

배 정 호

e-mail : jhbae83@pusan.ac.kr
2007년 부산대학교 정보컴퓨터공학(학사)
2009년 부산대학교 컴퓨터공학(석사)
2009년~현 재 부산대학교 컴퓨터공학과 박사과정
관심분야 : 소프트웨어공학, 테스팅, 리팩토링, 디자인 패턴 등

장 한 일

e-mail : hijang@suresofttech.com
2005년 부산대학교 정보컴퓨터공학(학사)
2007년 부산대학교 컴퓨터공학(석사)
2009년 현 재 슈어소프트테크 소프트웨어 시험자동화연구소 전임연구원
관심분야 : 프로그래밍언어 및 컴파일러, 프로그램 정적 분석, 프로그램 탐침, 소프트웨어 테스팅 등

이 윤 정

1995년 2월 부경대학교 전자계산학(학사)
1999년 2월 부경대학교 전산정보학(석사)
2008년 8월 부경대학교 전자계산학(박사)
2008년 9월~현 재 부산대학교 U-Port 사업단 박사후연구원
관심분야 : 얼굴 애니메이션, 웹 시각화

채 흥 석

e-mail : hschae@pusan.ac.kr

1994년 서울대 원자핵공학(학사)

1996년 한국과학기술원 전산학(석사)

2000년 한국과학기술원 전산학(박사)

2000년~2003년 ㈜ 동양시스템즈 기술연구소 선임연구원

2003년~2004년 한국과학기술원 전산학과 초빙교수

2009년 현 재 부산대학교 정보컴퓨터공학부 조교수

관심분야 : 객체지향 방법론, 소프트웨어 테스팅, 소프트웨어 메트릭, 소프트웨어 유지보수